ADVANCED MODELING AND SIMULATION SEMINAR SERIES

The Parallel Research Kernels, a tool for parallel systems investigations - Part II (https://github.com/ParRes/Kernels)

Rob Van der Wijngaart

Evangelos Georganas

Parallel system=hardware system+network stack+OS+parallel programming environment** (ProgEnv: programming model + API + compiler + runtime)*

Agenda

- Background/Motivation
- Particle-In-Cell kernel
- Adaptive Mesh Refinement kernel

Parallel Research Kernels (PRK)

Create test suite to study behavior or parallel systems

- Cover broad range of patterns found in real parallel applications
- Provide paper-and-pencil specification and generic reference implementations
- Ensure each kernel does some real work
- Keep kernels simple functionally
  - Easy porting to new runtimes/languages
  - Easy to understand by different domain scientists
  - Dominated by single feature, so convenient performance building block
- Parameterize kernels (problem size, iterations, # cores etc.)
- Include automatic verification test (analytical solution)
- Make sure kernels can be load balanced (enough concurrency)

Motivation to add kernels

- Initially PRK intended as architectural stress tests, not to compare runtimes
  - No insight into platform by studying fundamentally unbalanced load
  - Our solution: make kernels trivially statically load balanced
- However, exascale will require dynamic load balancing for mature workloads + system/network fluctuations
  - Balance load: ensure workers reach synchronization points at same time
  - Balance work: assign same amount of computational work to workers
- GOAL: Design and implement new kernels that:
  - Require dynamic load balancing at all system scales (algorithmic source)
  - Allow control of amount and frequency of workload adaptation
  - Have data dependencies, so load-balancing is non-trivial; improving load-balance usually increases communication
- Usage: Research vehicle to stress dynamic load-balancing capabilities of parallel runtimes

Algorithmic sources of dynamic load imbalance

Type I: Evolving mismatch between two (often distributed) data structures

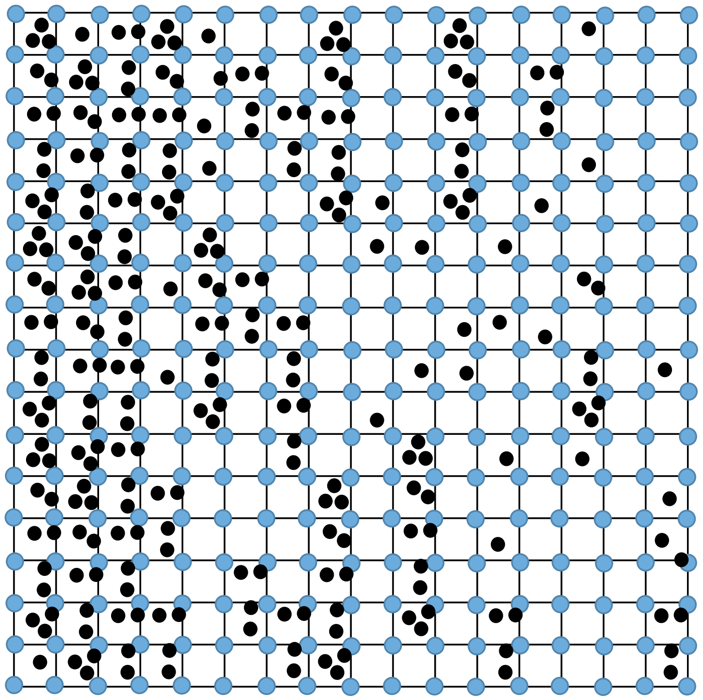
- Size of data structures constant
- Dependency between data structures
- No efficient static decomposition

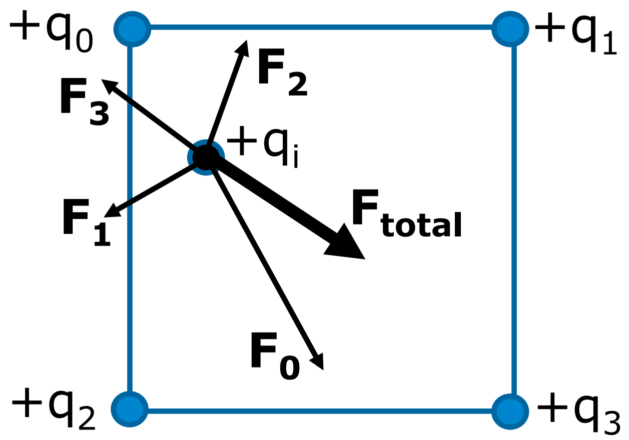
Type II: Work changes intermittently (dis/appears)

- Size of data structures changes
- New work depends on subset of existing data structure
- Equal distribution of new work among existing resources breaks locality + decreases granularity

Type I: Simple Particle-In-Cell (PIC)

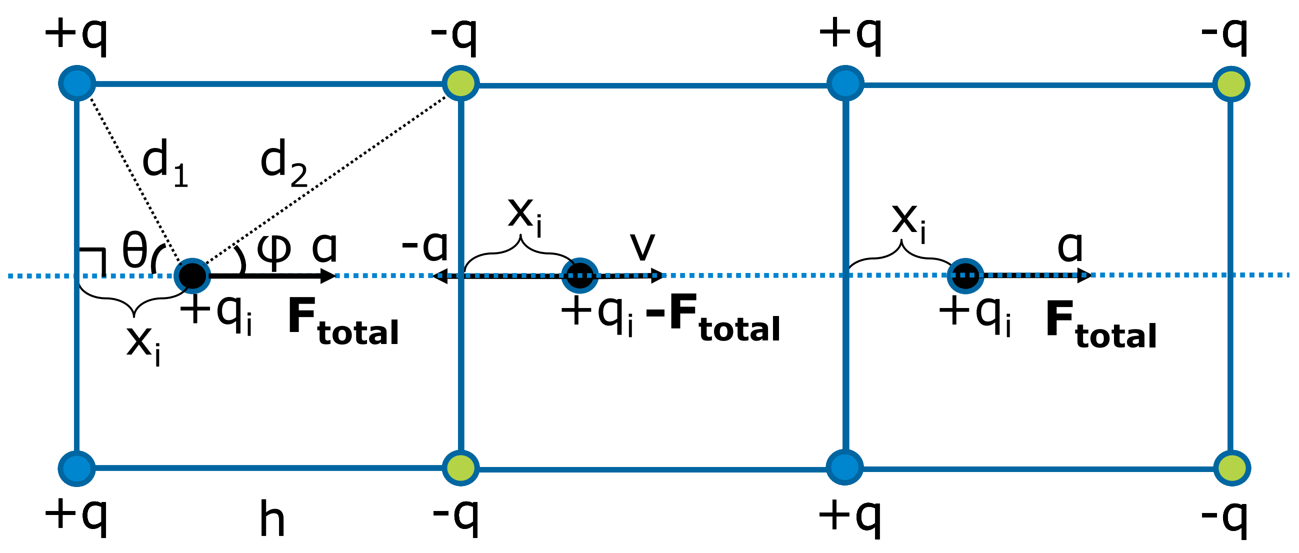
- 2D regular mesh with periodic boundaries
- Fixed charges at mesh points. No assumptions regarding charge distribution
- N particles, each with its own charge. No assumptions regarding particle distribution
- T discrete time steps of duration dt
- No interactions among moving particles
- Particles only interact with four charges at corners of containing cell
- Compute total Coulomb force $\mathbf{F}_{\mathbf{total}}$ on particle, corresponding to acceleration **a**: $\mathbf{a} = \mathbf{F}_{\mathbf{total}}/m$
- Given velocity **v**, position **x,** and acceleration **a** of a particle at time t, compute **v** and **x** at time t+dt:
  - $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v}\,dt + \frac{1}{2}\,\mathbf{a}\,dt^2$
  - $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{a}\,dt$





Initialization

- Alternate charges: Columns with even index have charge +q, columns with odd index have charge –q
- Put particles on horizontal axis of symmetry of cells
- Given the relative position x_i of particle i, assign charge q_i:
  $q_i = h / (q (\cos\theta/d_1^2 + \cos\varphi/d_2^2))$
- At time t+1 particle has shifted one cell, force reversed direction
- At time t+2 particle has shifted two cells, velocity and force identical to those at time t



Verification

Verification requirements/challenges:

1. Simple and short: don't take more time or memory than actual experiment.
2. Tight enough to catch even minor implementation errors, but not too tight (no bitwise accuracy)
3. Not relying on statistics (inaccurate for short/small experiments)

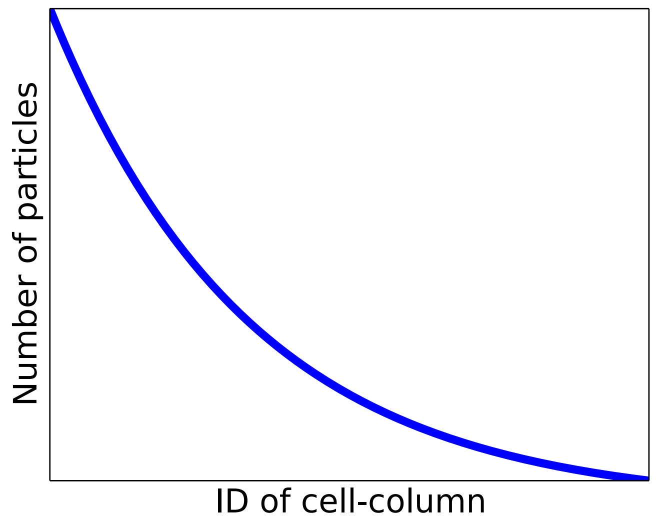
Actual verification

- Given initial coordinates (x_0 , y_0) and velocity $(0, z*h)$, final coordinates after T steps (modulo grid size) given by:
  - $x_T = x_0 + h T$
  - $y_T = y_0 + z h T$
- Assign unique id to each particle. Checksums of particle ids at and end of simulation must match.
- Verification test catches even a single miscalculated force or misplaced particle

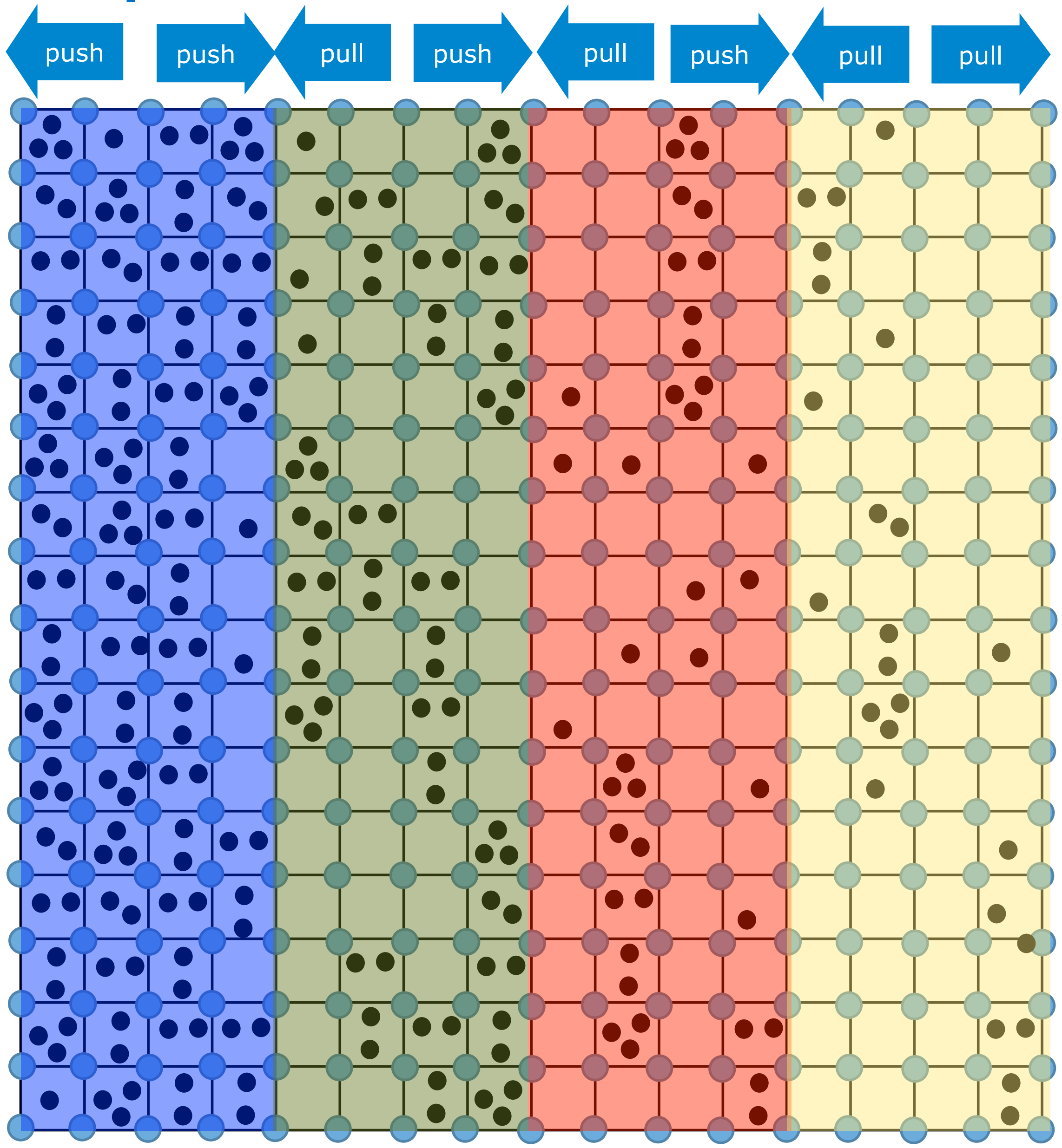
Load imbalance

- Nonuniform initial particle distribution makes simulation unbalanced; no static decomposition is efficient
- Uneven particle cloud moves through domain, requiring rebalancing
  - Frequency of needed rebalancing controlled by cloud speed

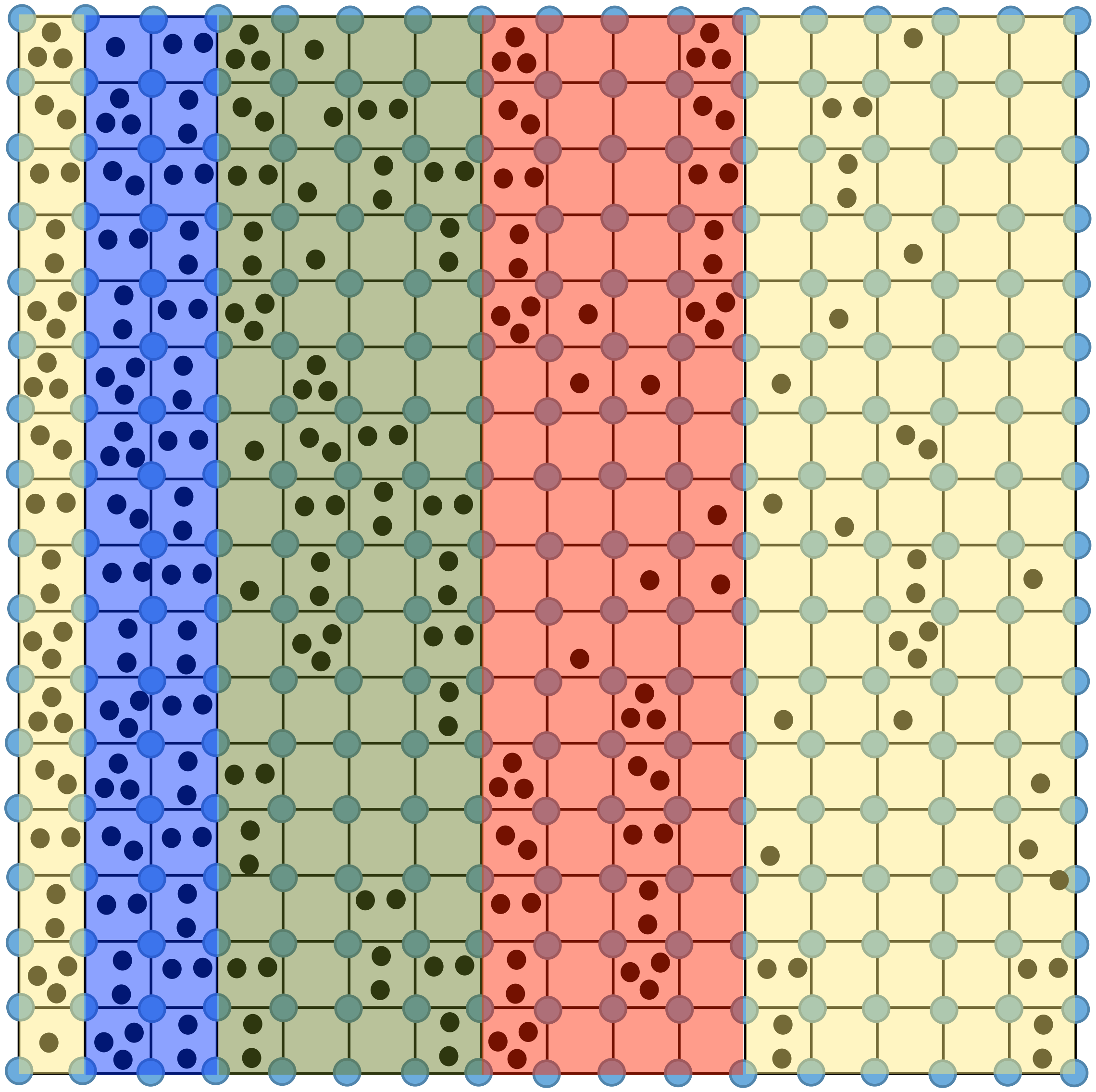
- **Example initial particle distribution:**
  - Column i of grid cells contains p(i) particles, where p(i) = A * r^i
  - Varying r makes distribution arbitrarily unbalanced
  - r =1 recovers uniform distribution



Example of 1D diffusion scheme

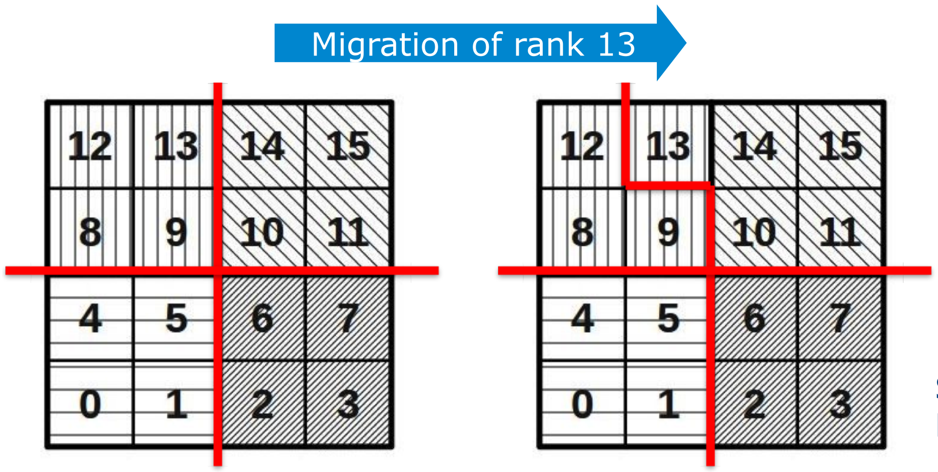


Example of 1D diffusion scheme



Runtime-based load balancing with Adaptive MPI (AMPI)

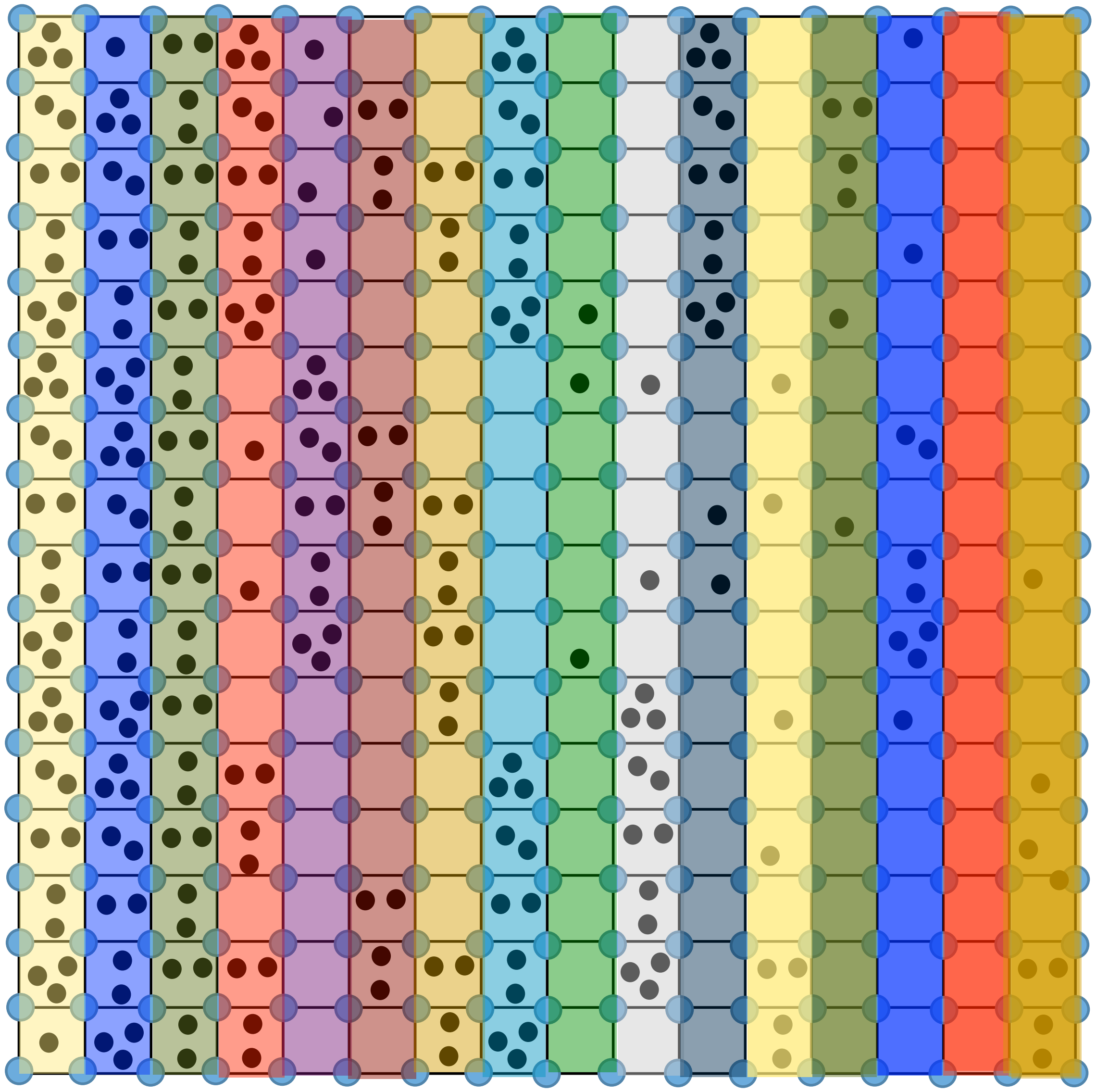
- AMPI: Multiple ranks (user-level threads) per process, typically one process per physical processor
- AMPI uses Charm++ scheduler for execution of ranks
- Approach: Over-decompose domain. AMPI migrates ranks across processors for load balancing
- Minimal changes to original static MPI implementation (serialization routines for dynamic data structures)



Source: ~~AMPI tutorial~~ Melania Trump

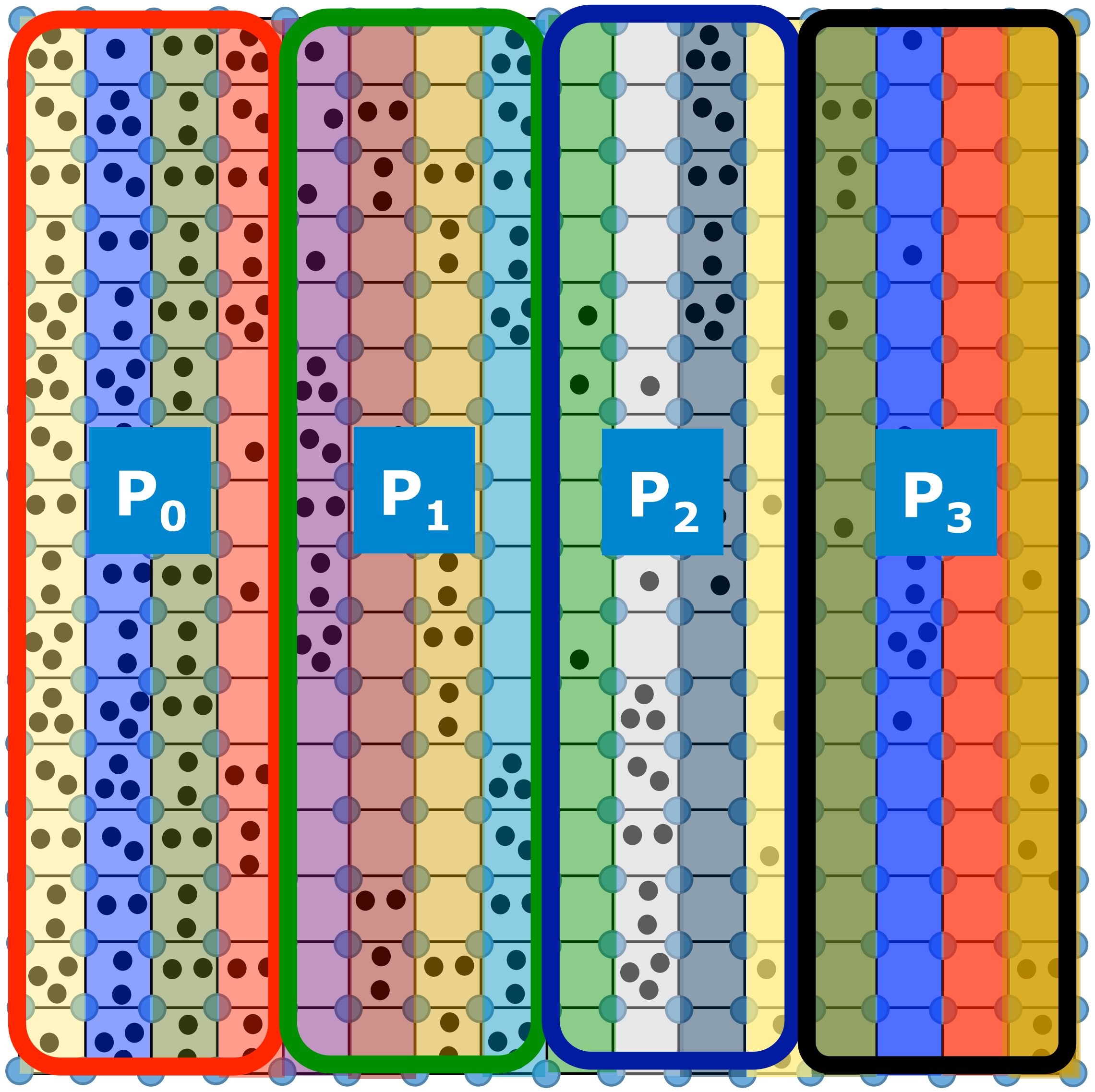
Example with Adaptive MPI

- Over-decompose domain to 16 ranks (on 4 processes)



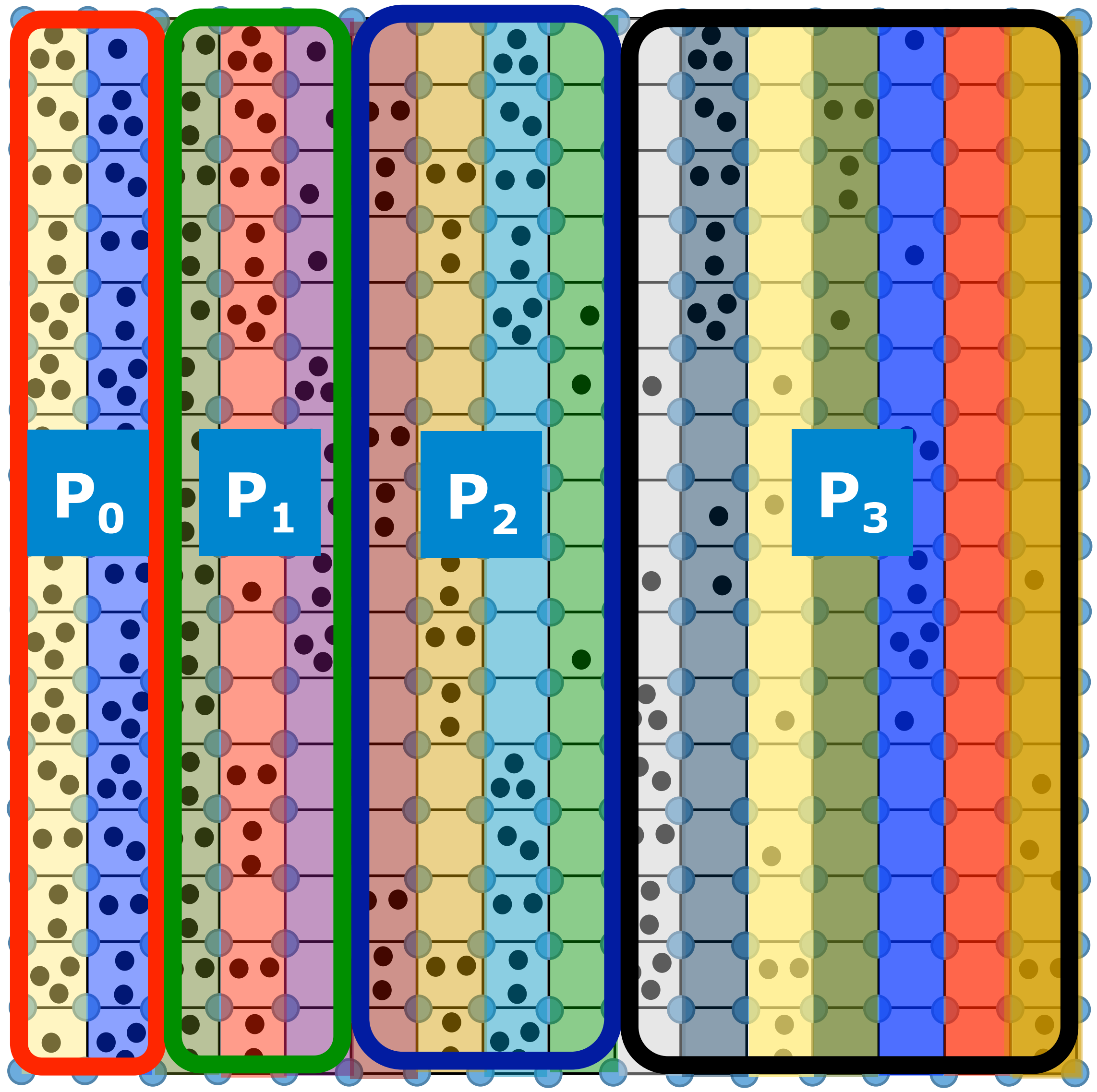
Example with Adaptive MPI

- Load balancer decides how to place/migrate ranks to processes



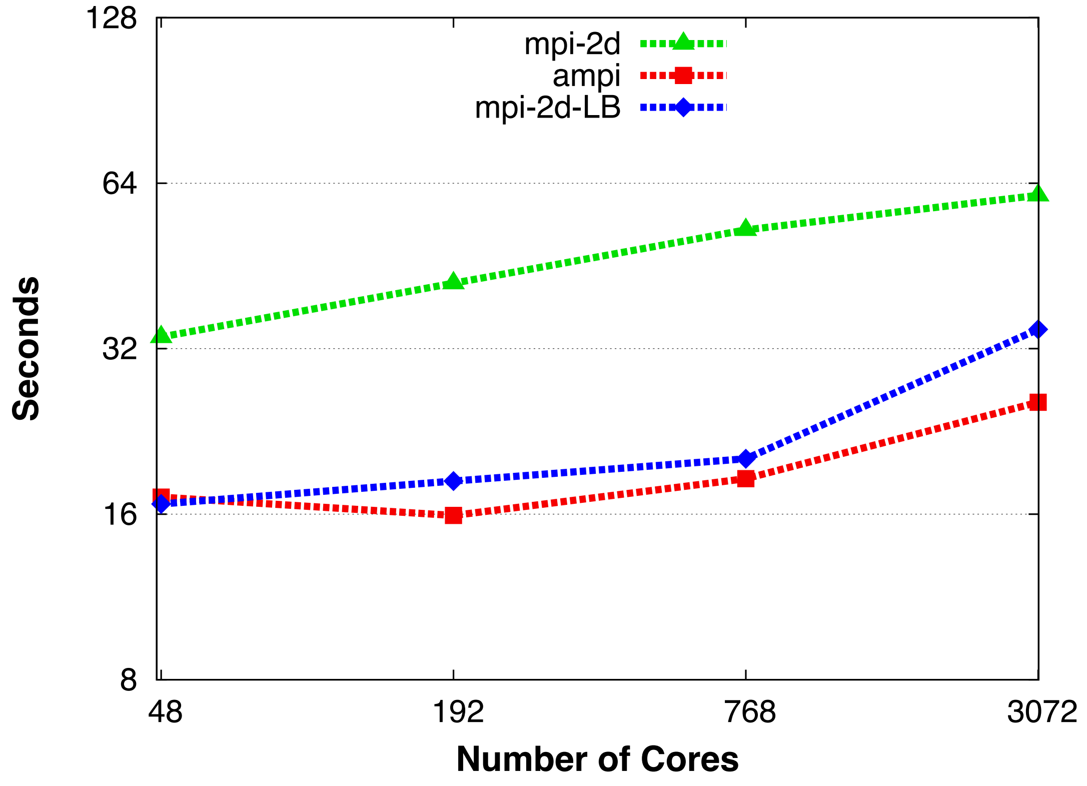
Example with Adaptive MPI

- Load balancer decides how to place/migrate ranks to processes



Distributed memory “weak” scaling

- 48 cores: $12K^2$ grid, 400K particles, 6K time steps; r= 0.999
- Increase number of cores and number of particles proportionally
- Load balancers: ampi: greedy, mpi-2d-LB: diffusion, mpi-2d: none



ampi:
- over-decomposition = 4
- frequency = 1/500, 1/250

mpi-2d-LB:
- quantum = 16 columns
- frequency = 1/4

Why Adaptive Mesh Refinement?

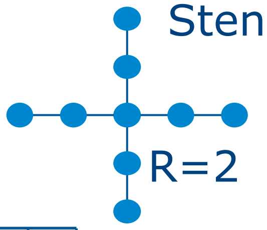
Limitations of PIC kernel (Type I dynamic load imbalance)

- Total amount of work constant; often work comes and goes in chunks: in situ visualization, AMR, computational steering, etc.
- Source of load imbalance is constant; no abrupt and/or unpredictable variations in load
- No analytical solution for *almost all* mesh sizes, initial particle placements, time steps, particle and grid charges
  - relies on infinite precision or cancelling of rounding errors
  - can become chaotic

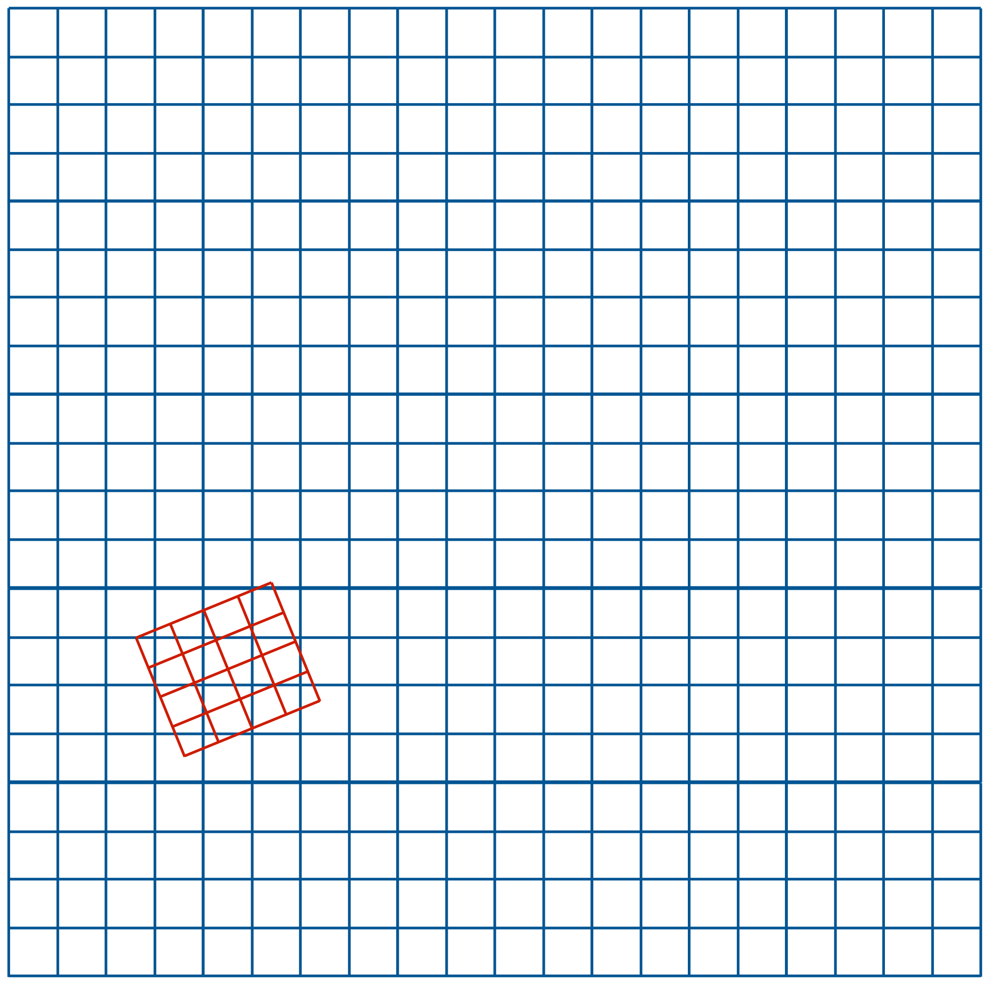
Fix: derive new kernel from Stencil PRK and AMR workloads (Type II)

- Has intermittent, abrupt introduction/removal of chunks of work
- Final solution continuous function of initial solution

Stencil kernel, BG with refinement grid

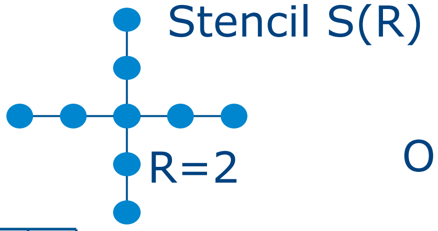


Out += S(R) In

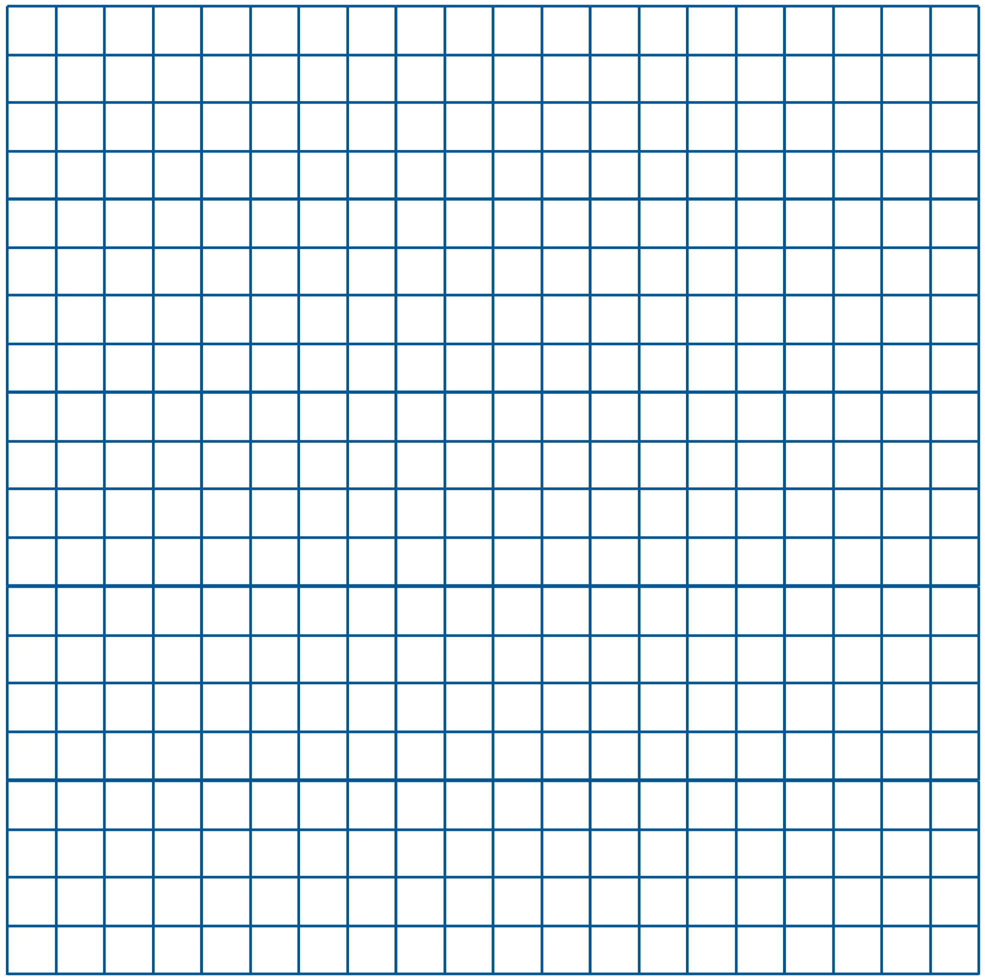


BG = Background Grid

Stencil kernel, BG with refinement grid



Out += S(R) In

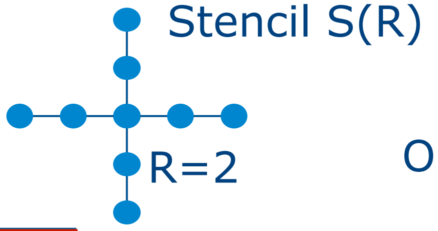


Refinement issues:

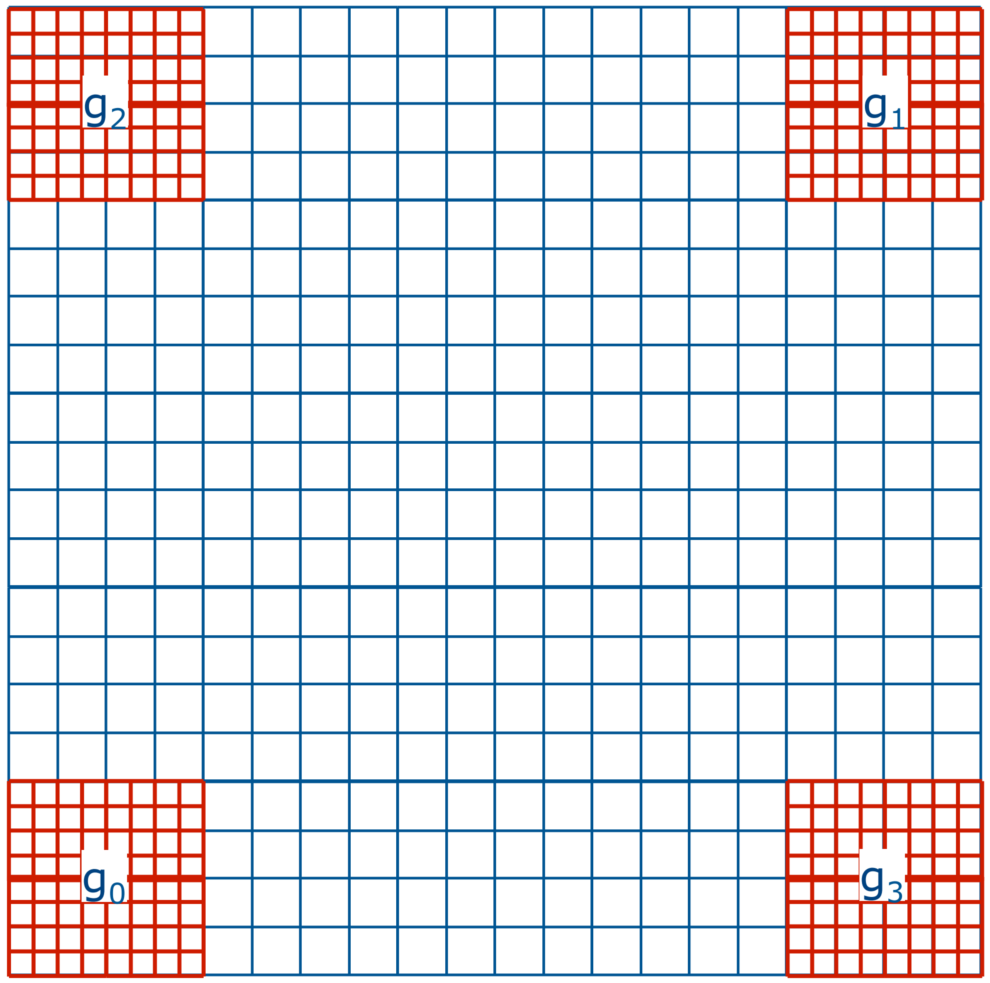
- How to ensure simple analytical solution?
- How should BG and refinement interact?
- How to avoid spending much time/effort on interpolation?
- How to avoid complicated path computation/bdry intersection?
- How to preserve refinement history?
- How to vary amount/ extent of refinement?
- How to vary frequency/ duration of refinement?

BG = Background Grid

Stencil kernel, BG with refinement grids



Out += S(R) In



Refinement scenario:

- Align BG and refinements
- Interpolate initial values on refinements from BG
- Keep refinements in place, but (de)activate cyclically
- Save state of all refinements
- Make refinements mesh size power-of-two of BG
- Define refinements in terms of BG cells
- Define refinements period/duration in terms of BG time steps
- Prescribe # iterations on refinements per BG iteration

BG = Background Grid

Reference implementations

- Application level dynamic load balancing (usually MPI)
  - Possible to distribute work of new refinement without global repartitioning?
- Runtime orchestrated dynamic load balancing (e.g. AMPI)
  - Employs static partitioning with over-decomposition

MPI (dynamic) load balancing; dumb and dumber—and dumbest

FINE_GRAIN: partition BG and refinements completely among all ranks

- Split BG evenly among all ranks
- When refinement appears, split evenly among all ranks

NO_TALK: minimize communication

- Split BG evenly among all ranks
- When refinement appears, split into pieces coinciding with BG partitioning and assign to same rank

HIGH_WATER: partition BG plus one refinement together statically

- Each rank receives exactly one subset of one of the grids at a time

NO_TALK_MULTI: same as NO_TALK, but over-decomposed

HIGH_WATER_MULTI: same as HIGH_WATER, but over-decomposed

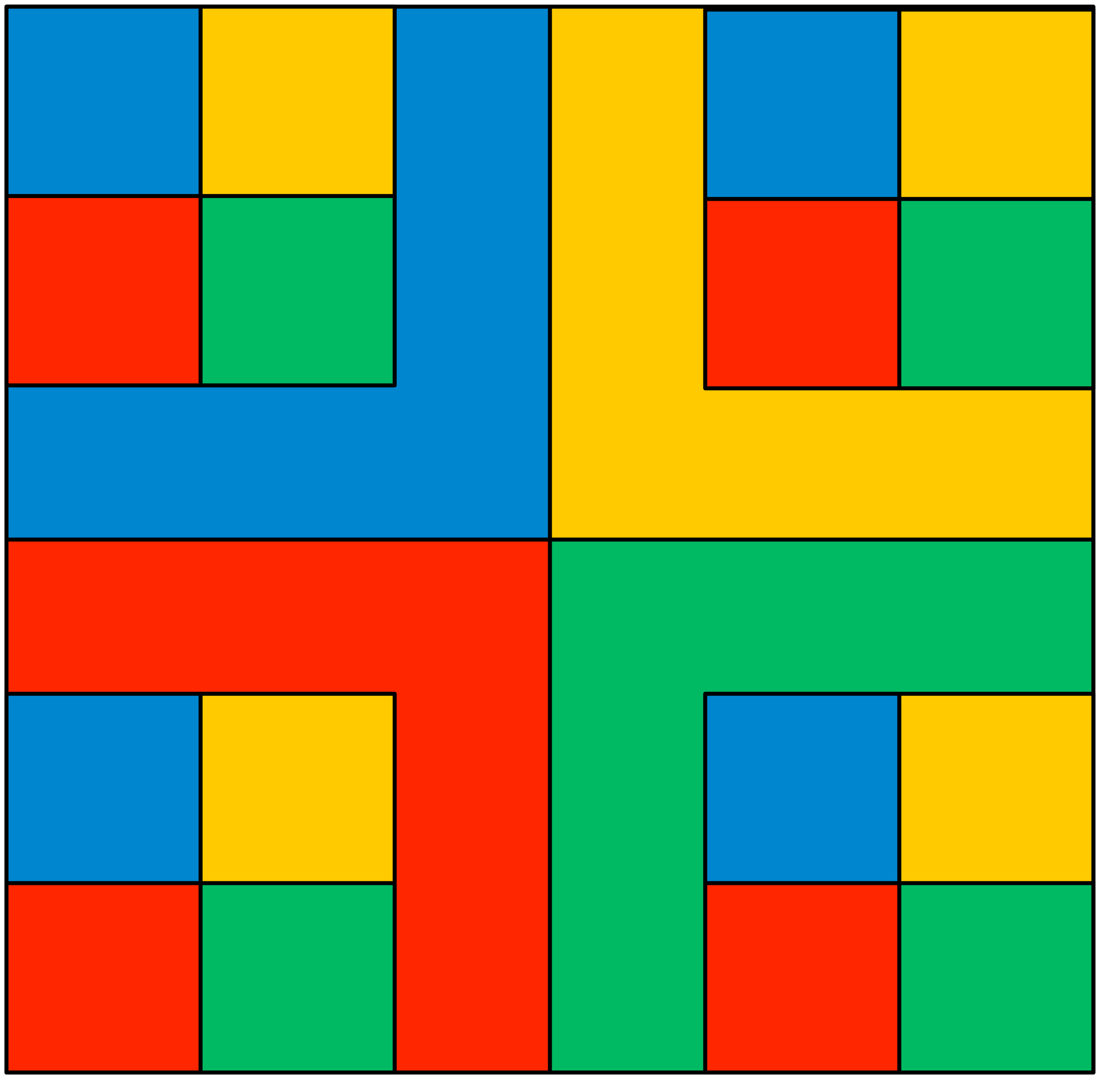
Static decomposition

NO_TALK_CORNER_CASE: shrink BG partitions towards refinement

AMNESIA: repartition each configuration from scratch (all-to-all)

Dynamic decomposition

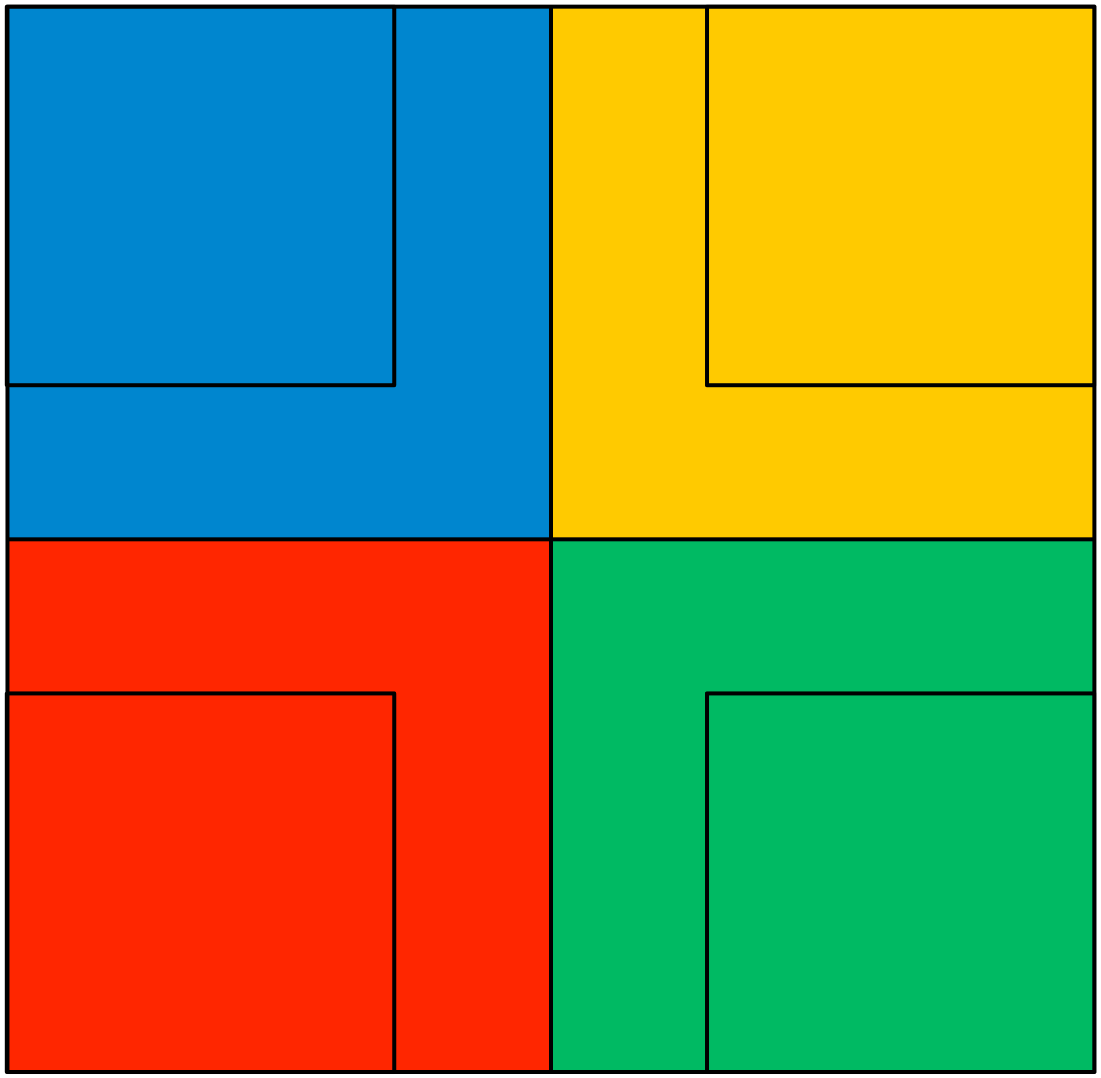
FINE_GRAIN



- Static decomposition
- Perfect load balance
- Good if BG work ≤ work on refinements, may become very fine-grain otherwise
- Poor inter-grid-level locality

Color = rank

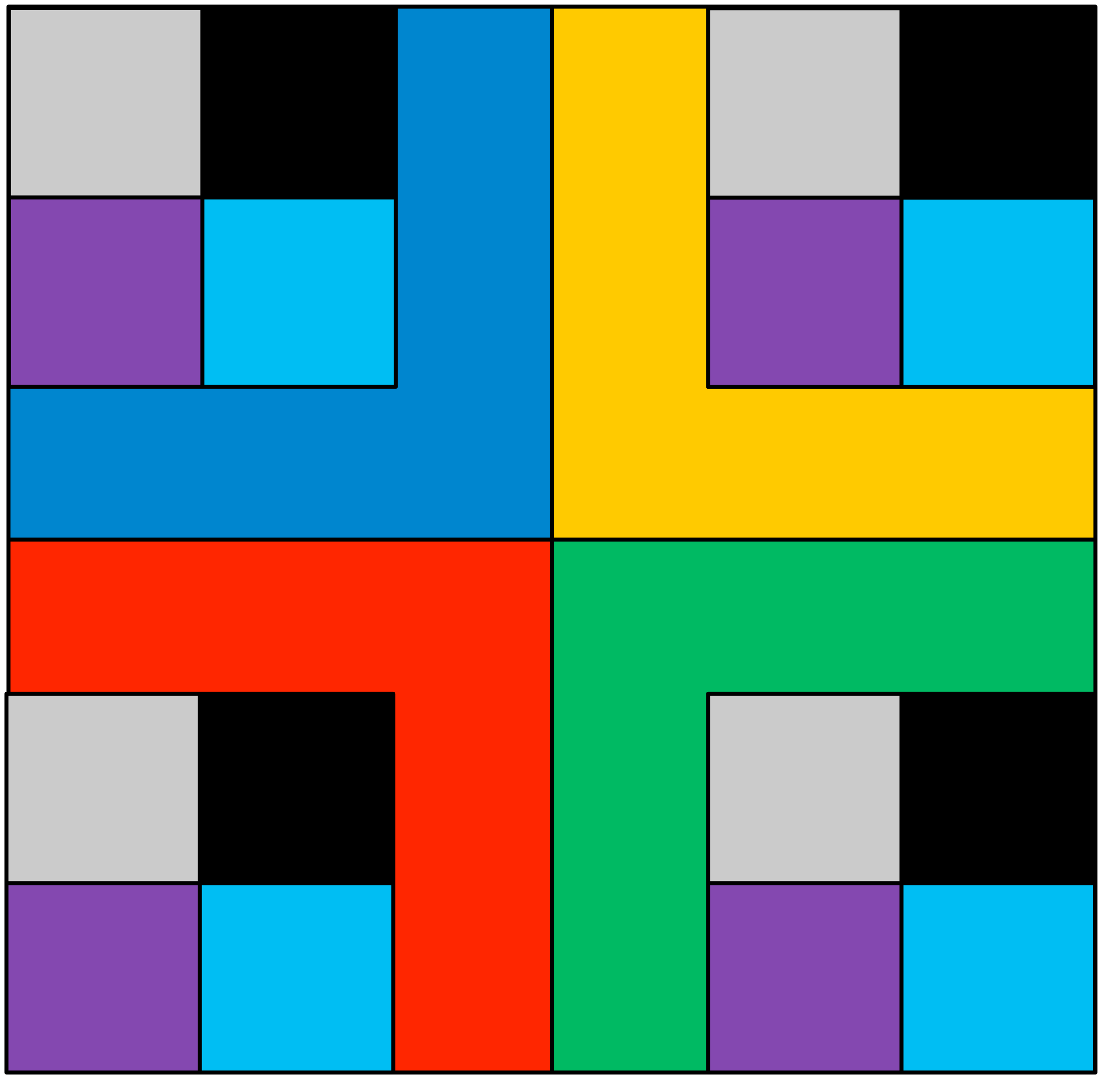
NO_TALK



- Static decomposition
- Perfect inter-grid-level glocality
- Perfect load balance between refinements
- Very poor load balance during refinements if work on refinements substantial
- Fine if work on refinements very small

Color = rank

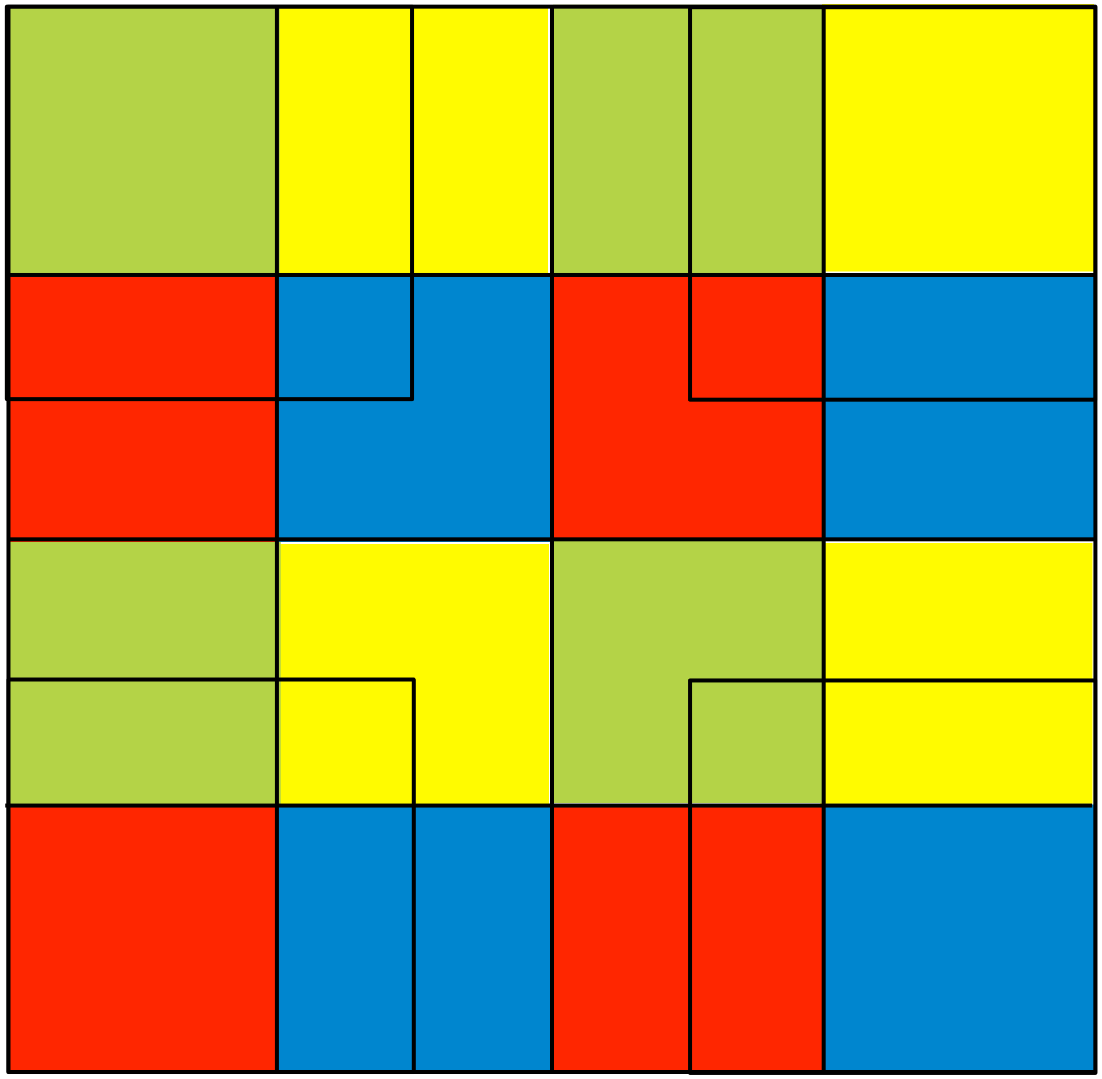
HIGH_WATER



- Static decomposition
- Perfect load balance during refinement
- Better granularity than FINE_GRAIN
- Very poor load balance between refinements, especially if BG work ≤ refinement work
- No inter-grid-level locality

Color = rank

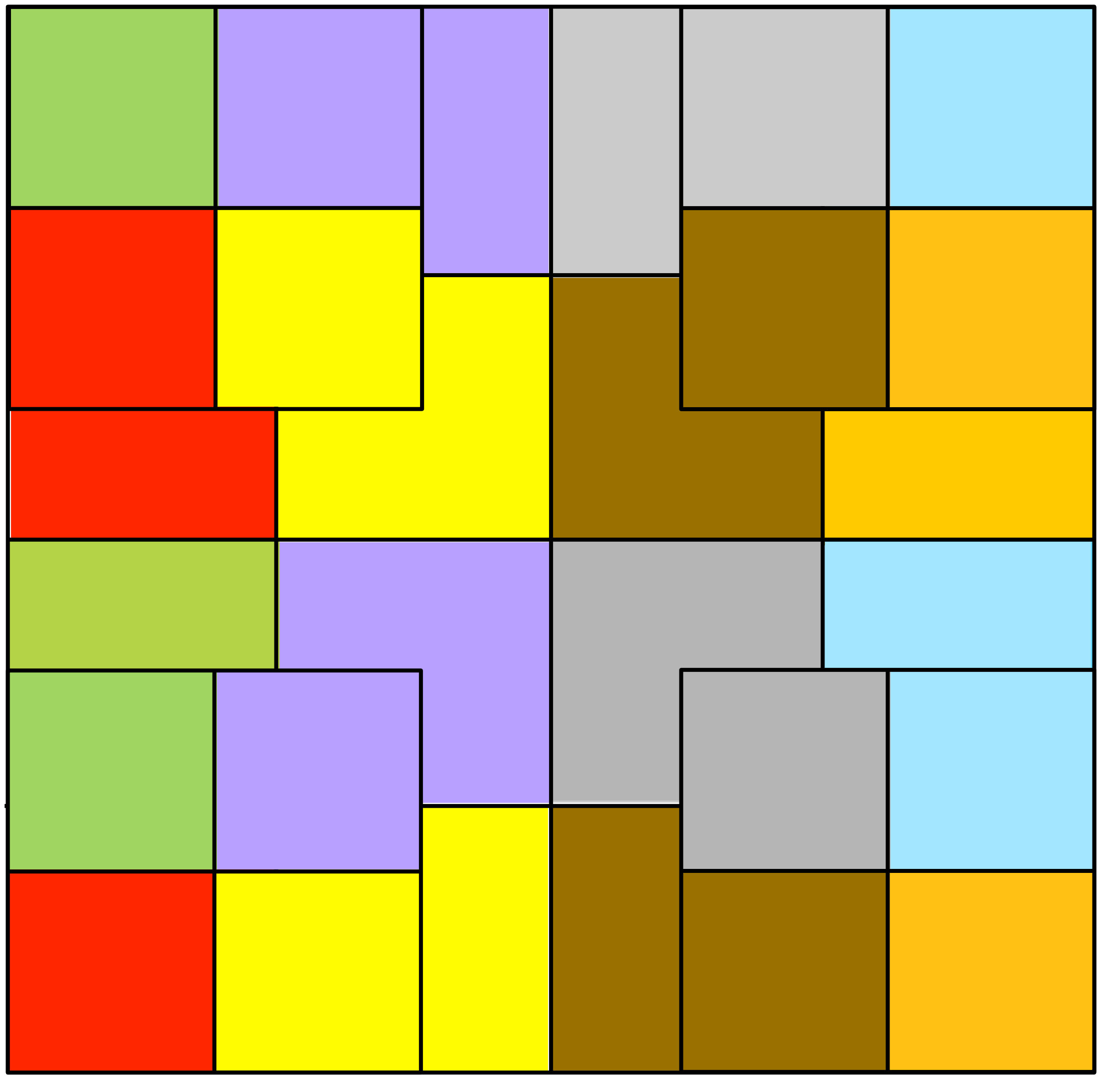
NO_TALK_MULTI



- Perfect inter-grid-level locality
- Perfect load balance between refinements
- Poor load balance during refinements if work on refinements substantial
- Finer-grained than NO_TALK

Color = worker

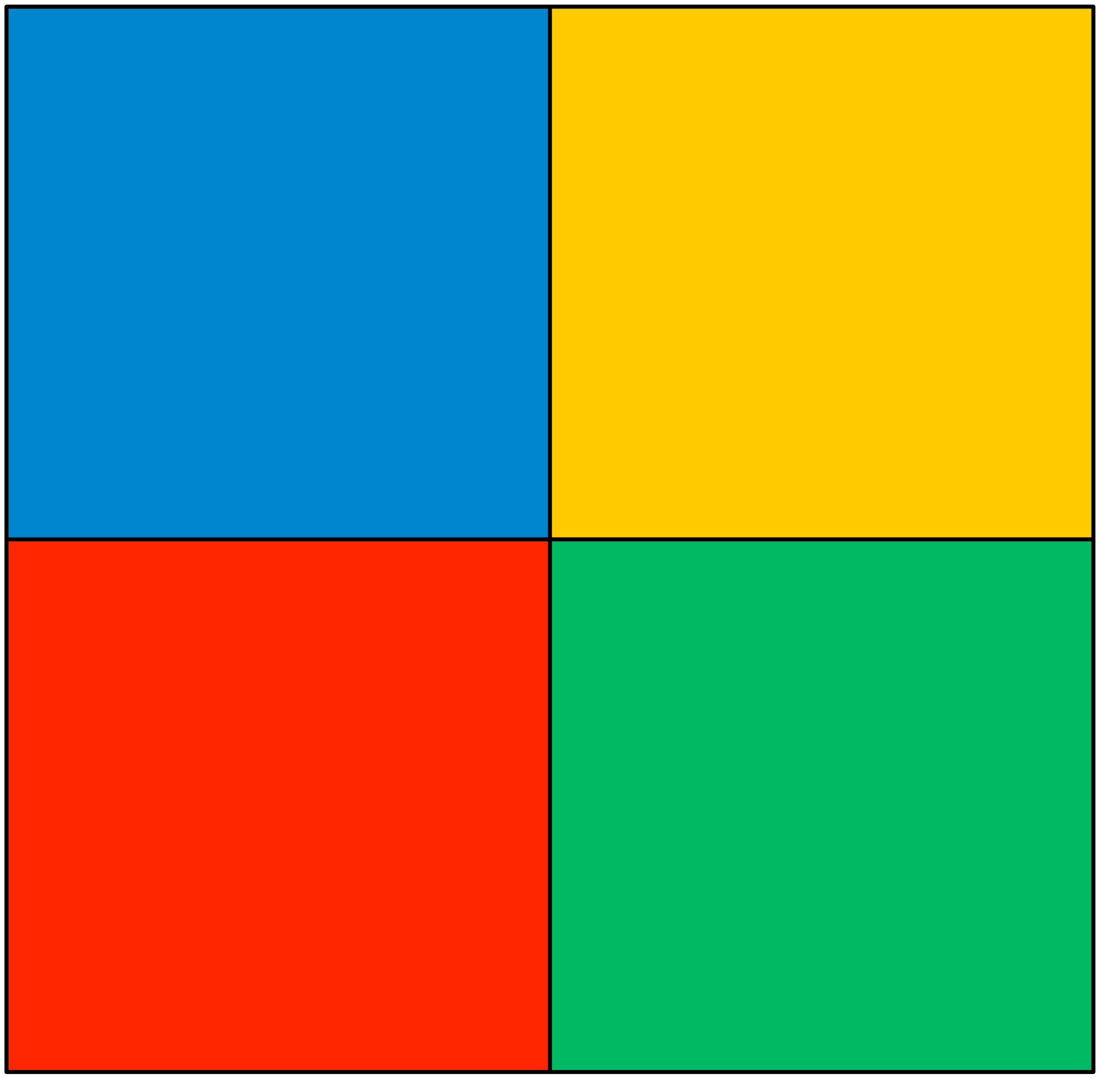
HIGH_WATER_MULTI



- Static decomposition
- Possibility for decent locality (depends on assignment quality)
- Poor load balance during refinements
- Perfect load balance between refinements
- Finer-grained than HIGH_WATER

Color = worker

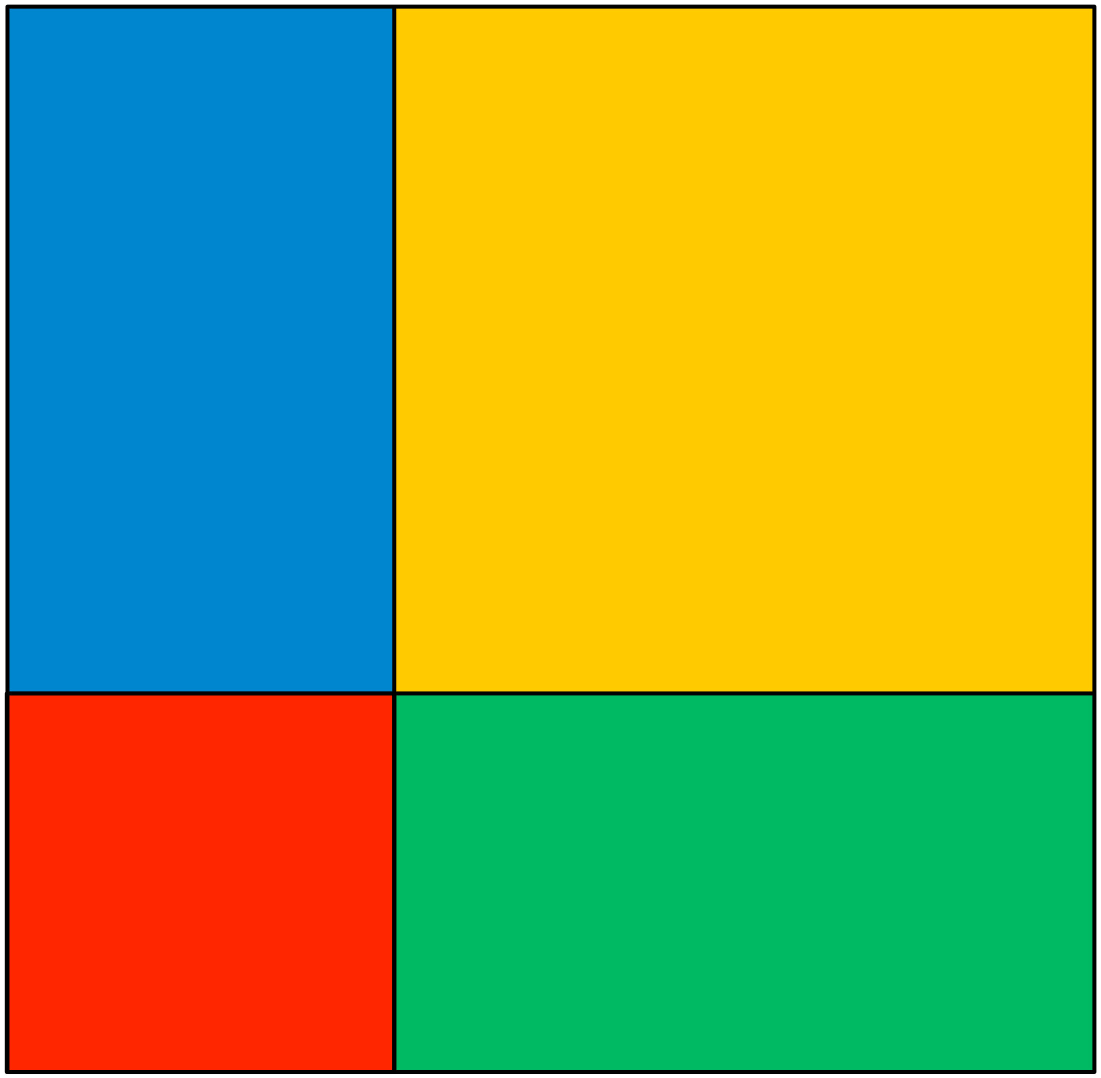
NO_TALK_CORNER_CASE



- Dynamic decomposition
- Perfect inter-grid-level locality
- Perfect load balance between refinements

Color = rank

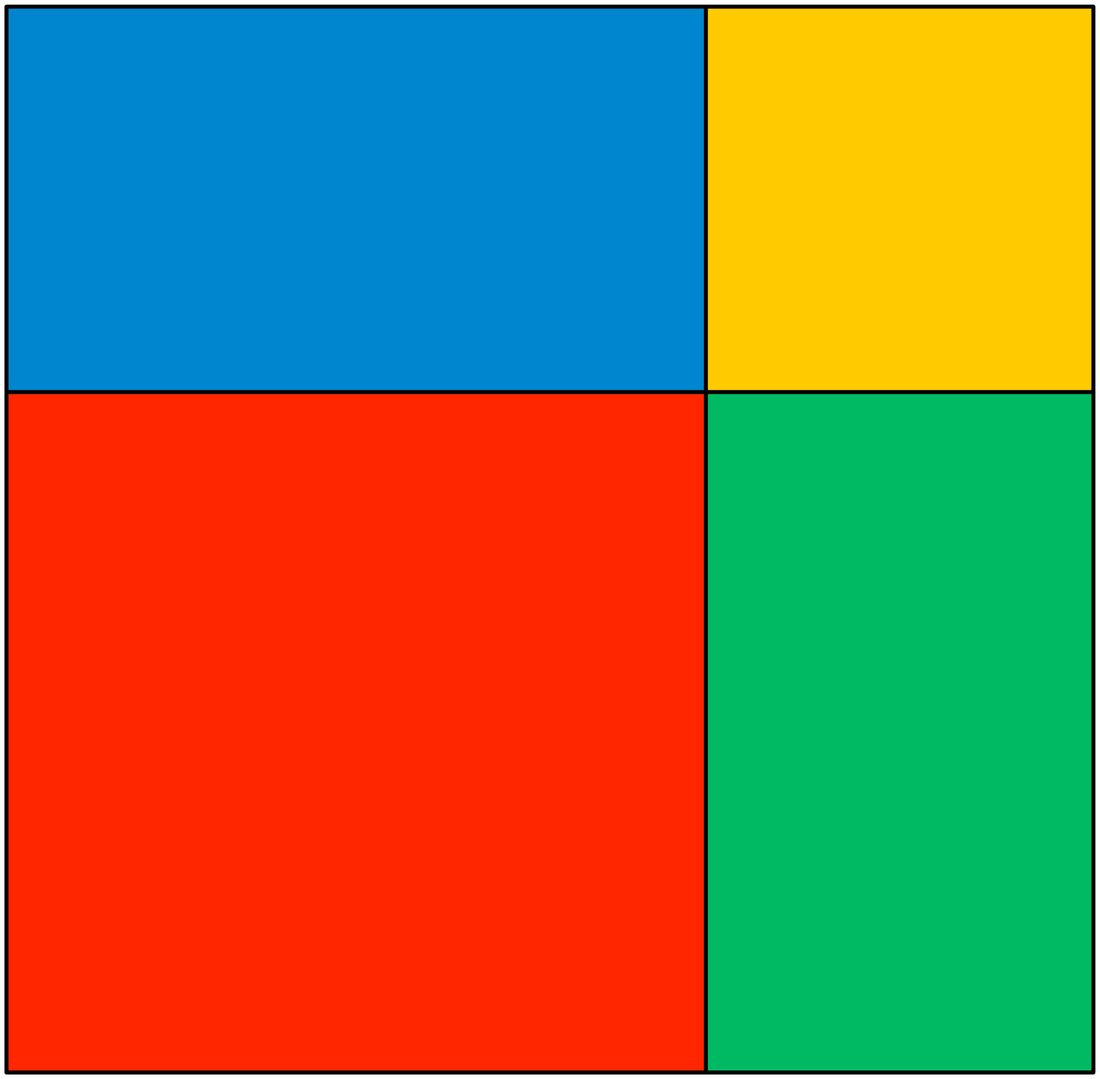
NO_TALK_CORNER_CASE



- Dynamic decomposition
- Perfect inter-grid-level locality
- Perfect load balance between refinements
- Better load balance during refinements than NO_TALK
- More communication to repartition BG

Color = rank

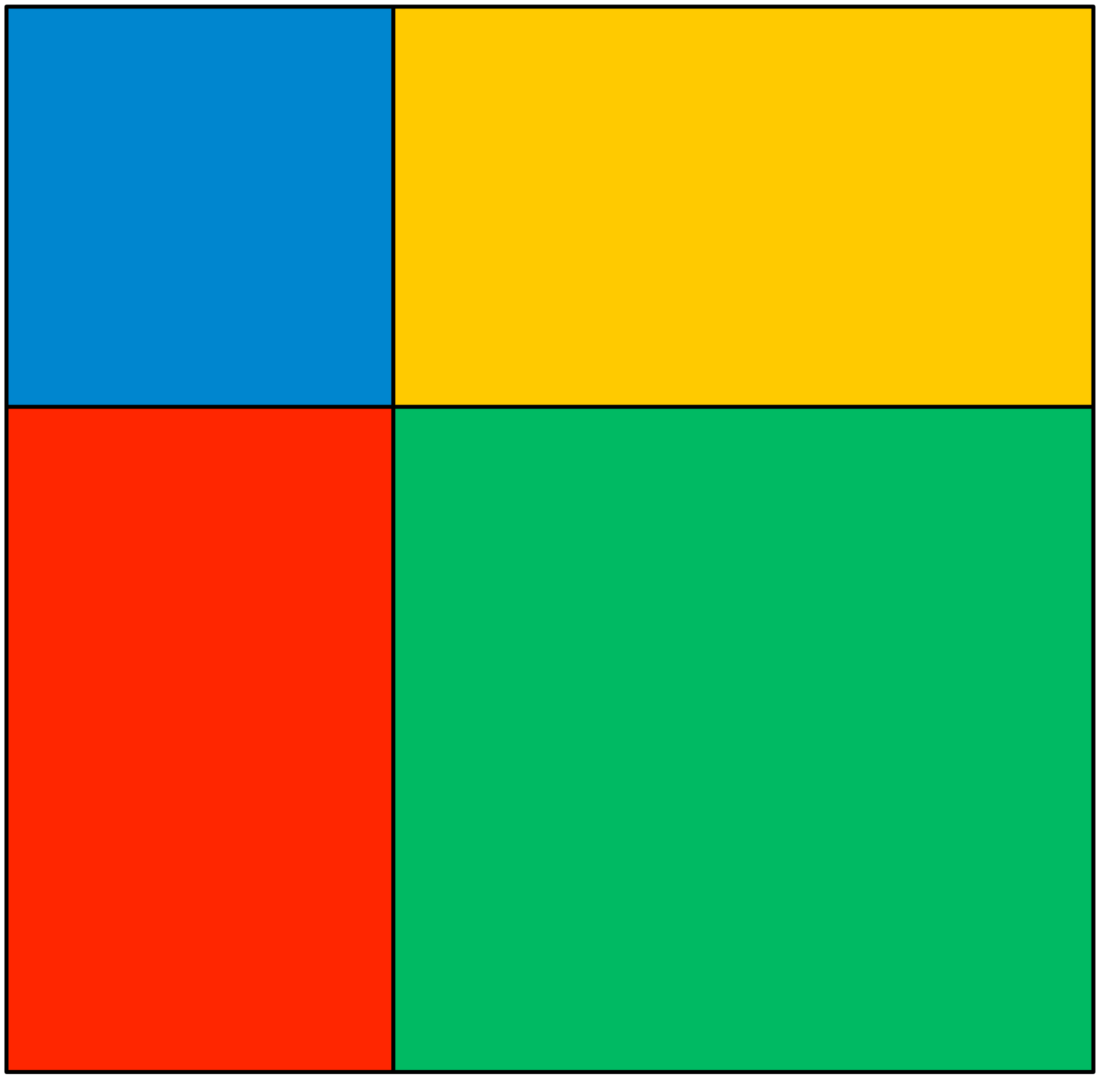
NO_TALK_CORNER_CASE



- Dynamic decomposition
- Perfect inter-grid-level locality
- Perfect load balance between refinements
- Better load balance during refinements than NO_TALK
- More communication to repartition BG

Color = rank

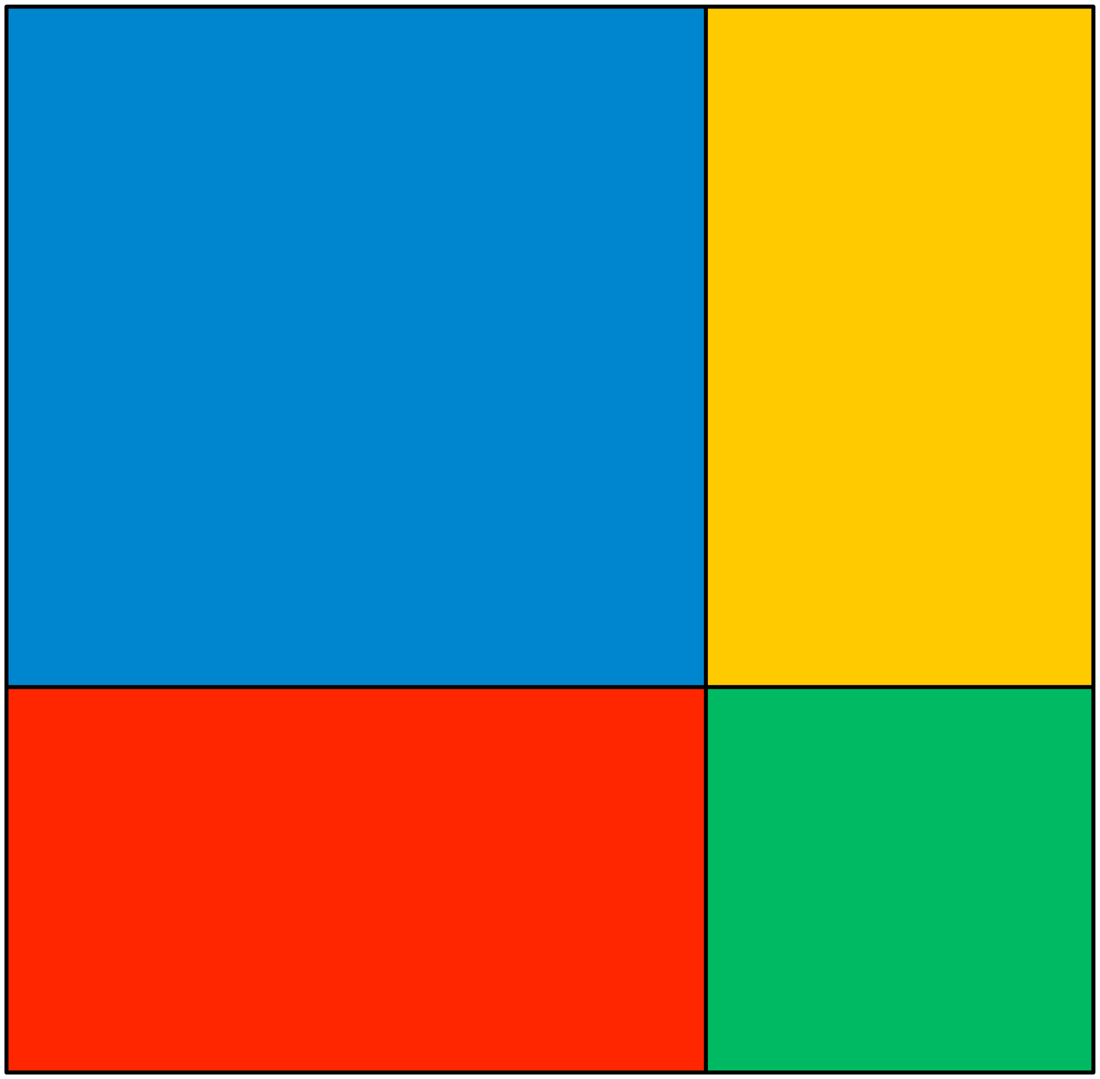
NO_TALK_CORNER_CASE



- Dynamic decomposition
- Perfect inter-grid-level locality
- Perfect load balance between refinements
- Better load balance during refinements than NO_TALK
- More communication to repartition BG

Color = rank

NO_TALK_CORNER_CASE



- Dynamic decomposition
- Perfect inter-grid-level locality
- Perfect load balance between refinements
- Better load balance during refinements than NO_TALK
- More communication to repartition BG

Color = rank

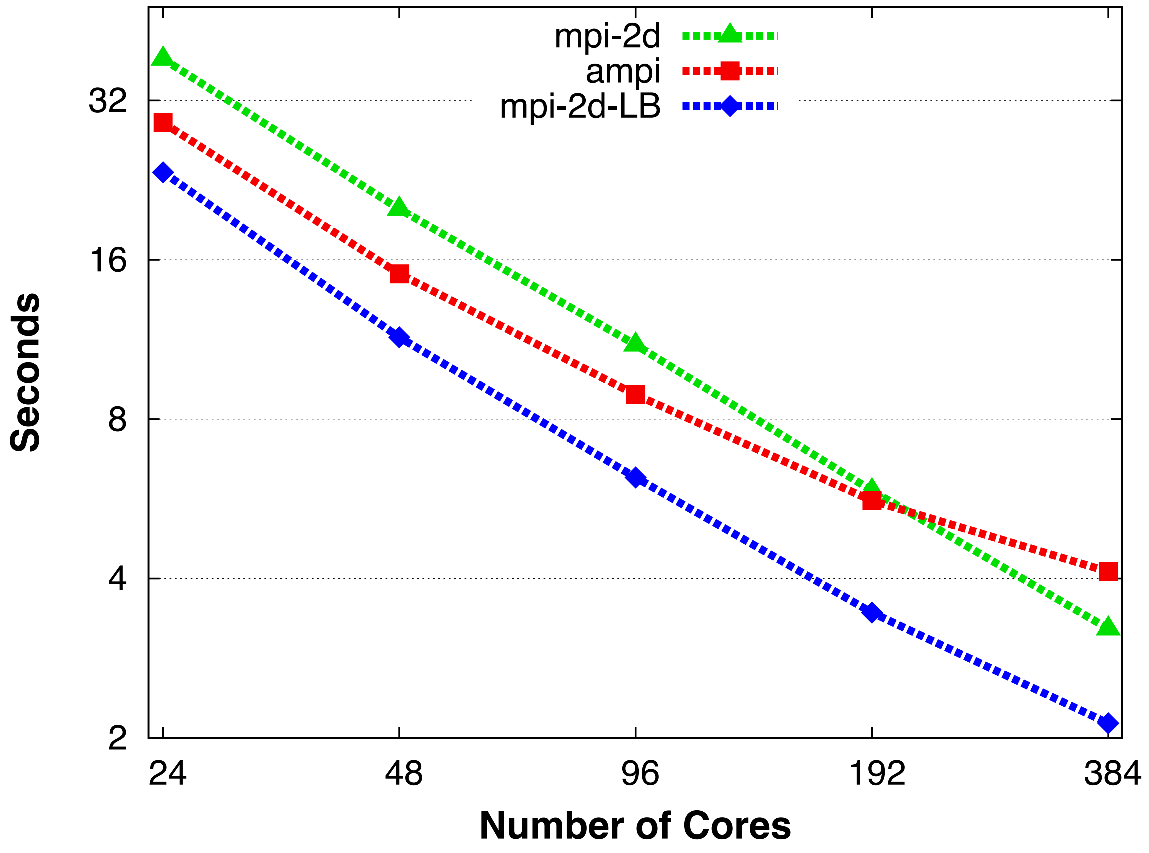
Conclusions

- PIC
  - Useful for comparing user-level and runtime-orchestrated dynamic load balancing of constant-work applications
- AMR
  - Tough case to parallelize using runtime-orchestrated dynamic load balancing
  - Is based on static partitioning with over-decomposition
  - When over-decomposition mitigates dynamic load imbalance, static mapping suffices
- Overall
  - AMR better proxy for localized system noise
  - Does not benefit from runtime-orchestrated dynamic load balancing
    - Local work increase cannot be absorbed locally unless workload already unbalanced before disturbance

Backup

Multiple nodes experiments

- Strong scaling experiment on Edison:
  - 2,999 x 2,999 grid, 600,000 particles (r = 0.999) and 6,000 time steps



AMR Specification details

Parameters

- T : total number of iterations (background grid)
- R: radius of difference stencil
- n: linear dimension of square background grid (n^2 points, mesh spacing is unity)
- r: refinement level (mesh size of refined grid is 2^{-r})
- k: linear dimension of refinement in terms of BG cells ($(k*2^r+1)^2$ points in each refinement)
- P : duration in terms of iterations on the BG of one full cycle of activation of one refinement until that of the next (*period*)
- D: duration in terms of iterations on the BG of activity on each refinement; $D \leq P$
- d: number of iterations on a refinement per iteration on the BG

AMR Specification details

(Re-)initialization

- $In[0](x,y) = c_x x + c_y y$
- $In_i[t] = \phi\,(In[t])$, with ϕ bi-linear interpolation (exact for linear field)

Update

- Increase In and In_i by constant after each stencil application

Verification

- S is numerical equivalent of ∇ (exact for linear field): $\nabla(c_x x + c_y y + \text{const}) = c_x + c_y$
- Count number of iterations η_i on $g_i \rightarrow Out_i[T](x,y) \equiv \eta_i * (c_x + c_y)$
- $Out[T](x,y) = T * (c_x + c_y)$
- $In[t](x,y) = c_x x + c_y y + t$, so: $\underline{In[T](x,y)} = (c_x + c_y)(n-1)/2 + T$
- Count number of updates ν_i on g_i since last interpolation at time $\theta_i \rightarrow \underline{In_i[T](x,y)} \equiv (c_x + c_y)*k/2 + \nu_i + f(corner_i) + \theta_i$

$corner_i$ = coordinates of bottom left corner point of g_i

Three example AMR scenarios

1. n=1000, 10 workers, r=1, k=100, P=3, D=1, d=1. Refinement has 1% of work of BG, lasts 1 iteration, then waits for 2 iterations until next refinement. OK to add refinement work to worker covering same part of BG (~10% load imbalance)
2. n=1000, 100 workers, r=1, k=100, P=3, D=1, d=1. Not OK to add refinement work to worker covering same part of BG (100% load imbalance). Rapid (dis)appearance requires frequent load balancing
3. n=1000, 100 workers, r=4, k=6, P=30, D = 10, d = 5. Refinements ≈number of grid points as in scenario 1, but cover much smaller fraction of the BG; activated 10x slower than in that case, persist 50x longer, so automatic load balancing may respond effectively to changes in load